**os\_lab3\_32207522\_양상훈**

**구현(코드, 실행 결과)**

**코드**

**ftl\_impl.h**

#include "ftl.h"

#ifndef FTL\_IMPL\_H

#define FTL\_IMPL\_H

class GreedyFTL : public FlashTranslationLayer {

private:

// 멤버 변수 추가 선언 가능

bool in\_gc = false;

public:

// 생성자

GreedyFTL(int total\_blocks, int block\_size) : FlashTranslationLayer(total\_blocks, block\_size) {

name = "GreedyFTL";

}

// 소멸자

~GreedyFTL() {}

// 멤버 함수 추가 선언 가능

void garbageCollect() override;

void writePage(int logicalPage, int data) override;

void readPage(int logicalPage) override;

// 사용 중인 블록 중 valid 페이지가 2개 이하인 블록이 있는지 확인

bool shouldTriggerGC() {

// 모든 블록을 순회하면서 GC 트리거 조건 확인

for (const auto& blk : blocks) {

if (blk.is\_free) continue; // free 블록은 제외

// 사용 중인 블록에서 valid 페이지가 2개 이하인 경우

if (blk.valid\_page\_cnt <= 2) {

return true;

}

}

return false;

}

// 헬퍼 함수 - 0번 블록부터 순차적으로 탐색

int getFreeBlockIndex() {

for (int i = 0; i < total\_blocks; i++) {

// 비어있는 블록을 찾으면 해당 인덱스 반환

if (blocks[i].is\_free) return i;

}

// 모든 블록이 사용 중이면 -1 반환

return -1;

}

// GC 수행 가능한 희생자 블록이 존재하는지 확인

bool hasVictimBlock() {

// 모든 블록을 순회하면서 희생자 블록 조건 확인

for (int i = 0; i < total\_blocks; ++i) {

Block& blk = blocks[i];

// 비어있는 블록은 희생자가 될 수 없으므로 건너뛰기

if (blk.is\_free) continue;

// valid 페이지가 2개 초과인 블록은 GC 대상에서 제외

if (blk.valid\_page\_cnt > 2) continue;

// invalid 페이지가 있는 블록이면 희생자 블록으로 적합

if (blk.invalid\_page\_cnt > 0) return true;

}

// 조건을 만족하는 희생자 블록이 존재하지 않음

return false;

}

};

class CostBenefitFTL : public FlashTranslationLayer {

private:

// 멤버 변수 추가 선언 가능

bool in\_gc = false;

public:

// 생성자

CostBenefitFTL(int total\_blocks, int block\_size) : FlashTranslationLayer(total\_blocks, block\_size) {

name = "CostBenefitFTL";

}

// 소멸자

~CostBenefitFTL() {}

// 멤버 함수 추가 선언 가능

void garbageCollect() override;

void writePage(int logicalPage, int data) override;

void readPage(int logicalPage) override;

// 사용 중인 블록 중 valid 페이지가 2개 이하인 블록이 있는지 확인

bool shouldTriggerGC() {

// 모든 블록을 순회하면서 GC 트리거 조건 확인

for (const auto& blk : blocks) {

if (blk.is\_free) continue; // free 블록은 제외

// 사용 중인 블록에서 valid 페이지가 2개 이하인 경우

if (blk.valid\_page\_cnt <= 2) {

return true;

}

}

return false;

}

};

#endif // FTL\_IMPL\_H

**ftl\_impl.cpp**

/\*

\* DKU Operating System Lab

\* Lab3 (Flash Translation Layer)

\* Student id : 32207522

\* Student name : 양상훈

\* Date : 2025/06/13

\*/

#include "ftl\_impl.h"

// GreedyFTL의 가비지 컬렉션 구현

void GreedyFTL::garbageCollect() {

// 이미 GC가 진행 중이면 중복 실행 방지

if (in\_gc) return;

// GC 시작 플래그 설정

in\_gc = true;

// 희생자 블록을 찾기 위한 변수들 초기화

int victim\_block = -1;

int max\_invalid = -1;

// 모든 블록을 순회하면서 최적의 희생자 블록 찾기

for (int i = 0; i < total\_blocks; ++i) {

Block& blk = blocks[i];

// 이미 비어있는 블록은 건너뛰기

if (blk.is\_free) continue;

// valid 페이지가 2개 초과인 블록은 GC 대상에서 제외

if (blk.valid\_page\_cnt > 2) continue;

// invalid 페이지가 없는 블록은 GC 의미 없음

if (blk.invalid\_page\_cnt <= 0) continue;

// 가장 많은 invalid 페이지를 가진 블록을 희생자로 선택 (Greedy 정책)

if (blk.invalid\_page\_cnt > max\_invalid) {

max\_invalid = blk.invalid\_page\_cnt;

victim\_block = i;

}

}

// 적절한 희생자 블록을 찾지 못한 경우

if (victim\_block == -1) {

// 모든 페이지가 Valid한 경우 오류 메시지 출력

std::cout << "Error: No space available for garbage collection\n";

in\_gc = false;

return;

}

// 선택된 블록의 GC 수행 횟수 증가

blocks[victim\_block].gc\_cnt++;

// 희생자 블록의 모든 페이지를 순회

for (int i = 0; i < block\_size; ++i) {

Page& page = blocks[victim\_block].pages[i];

// 유효한 페이지만 다른 곳으로 이동

if (page.state == VALID) {

// 페이지의 논리 주소와 데이터 백업

int lpn = page.logical\_page\_num;

int data = page.data;

// 기존 물리 주소 백업하고 L2P 테이블에서 임시 제거

int prev\_ppn = L2P[lpn];

L2P[lpn] = -1;

// GC 플래그 상태 백업

bool prev\_gc = in\_gc;

in\_gc = true;

// 유효한 데이터를 새로운 위치에 쓰기

writePage(lpn, data);

// writePage가 실패한 경우 롤백 처리

if (L2P[lpn] == -1) {

// 원래 페이지 상태 복구

page.state = VALID;

blocks[victim\_block].valid\_page\_cnt++;

blocks[victim\_block].invalid\_page\_cnt--;

// L2P 테이블도 원래 상태로 복구

L2P[lpn] = prev\_ppn;

}

// GC 플래그 복원

in\_gc = prev\_gc;

// WAF 계산에서 GC로 인한 쓰기는 제외 (double 타입으로 처리)

if (L2P[lpn] != -1) {

total\_logical\_writes -= 1.0;

}

}

}

// 블록 초기화 전에 GC 횟수 백업

int gc\_count = blocks[victim\_block].gc\_cnt;

// 블록을 완전히 초기화

blocks[victim\_block] = Block();

// GC 횟수 복원

blocks[victim\_block].gc\_cnt = gc\_count;

// 페이지 배열 재할당

blocks[victim\_block].pages.resize(block\_size);

// GC 완료 플래그 해제

in\_gc = false;

}

// GreedyFTL의 페이지 쓰기 구현

void GreedyFTL::writePage(int logicalPage, int data) {

// 논리 페이지 번호 유효성 검사

if (logicalPage < 0 || logicalPage >= static\_cast<int>(L2P.size())) return;

// 총 논리적 쓰기 횟수 증가 (double 타입으로 처리)

total\_logical\_writes += 1.0;

// GC 중이 아니고 GC 트리거 조건을 만족하며 희생자 블록이 있으면 GC 수행

if (!in\_gc && shouldTriggerGC() && hasVictimBlock()) {

garbageCollect();

}

// 기존에 매핑된 물리 페이지가 있으면 무효화

int old\_ppn = L2P[logicalPage];

if (old\_ppn != -1) {

// 물리 주소에서 블록 번호와 오프셋 계산

int old\_block = old\_ppn / block\_size;

int old\_offset = old\_ppn % block\_size;

// 블록과 오프셋 유효성 검사

if (old\_block >= 0 && old\_block < total\_blocks &&

old\_offset >= 0 && old\_offset < block\_size) {

Page& old\_page = blocks[old\_block].pages[old\_offset];

// 유효한 페이지를 무효로 변경

if (old\_page.state == VALID) {

old\_page.state = INVALID;

blocks[old\_block].valid\_page\_cnt--;

blocks[old\_block].invalid\_page\_cnt++;

}

}

}

// 활성 블록이 없거나 꽉 찬 경우 새 블록 할당

if (active\_block == -1 || active\_offset >= block\_size) {

// 헬퍼 함수를 사용해 자유 블록 찾기

int free\_block = getFreeBlockIndex();

if (free\_block != -1) {

// 새 활성 블록 설정

active\_block = free\_block;

active\_offset = 0;

// 블록을 사용 중으로 표시

blocks[active\_block].is\_free = false;

// 페이지 배열 초기화

blocks[active\_block].pages.resize(block\_size);

} else if (!in\_gc) {

// 자유 블록이 없고 GC 중이 아니면 GC 수행

garbageCollect();

// GC 후 다시 자유 블록 찾기

free\_block = getFreeBlockIndex();

if (free\_block != -1) {

active\_block = free\_block;

active\_offset = 0;

blocks[active\_block].is\_free = false;

blocks[active\_block].pages.resize(block\_size);

}

}

// 여전히 활성 블록을 할당받지 못한 경우

if (active\_block == -1) {

// 실패한 쓰기에 대한 카운터 복구

total\_logical\_writes -= 1.0;

return;

}

}

// 새 페이지에 데이터 쓰기

int ppn = active\_block \* block\_size + active\_offset;

Page& new\_page = blocks[active\_block].pages[active\_offset];

// 페이지 정보 설정

new\_page.logical\_page\_num = logicalPage;

new\_page.state = VALID;

new\_page.data = data;

// 블록의 유효 페이지 수 증가

blocks[active\_block].valid\_page\_cnt++;

// 마지막 쓰기 시간 기록 (double을 int로 캐스팅)

blocks[active\_block].last\_write\_time = static\_cast<int>(total\_logical\_writes);

// L2P 테이블 업데이트

L2P[logicalPage] = ppn;

// 다음 쓰기를 위한 오프셋 증가

active\_offset++;

// 총 물리적 쓰기 횟수 증가

total\_physical\_writes += 1.0;

}

// GreedyFTL의 페이지 읽기 구현

void GreedyFTL::readPage(int logicalPage) {

// 논리 페이지 번호 유효성 검사

if (logicalPage < 0 || logicalPage >= static\_cast<int>(L2P.size())) return;

// L2P 테이블에서 물리 주소 찾기

int ppn = L2P[logicalPage];

// 매핑되지 않은 페이지는 읽을 수 없음

if (ppn == -1) return;

// 물리 주소에서 블록과 오프셋 계산

int blk = ppn / block\_size;

int offset = ppn % block\_size;

// 블록과 오프셋 유효성 검사

if (blk < 0 || blk >= total\_blocks || offset < 0 || offset >= block\_size) return;

// 해당 페이지 참조

Page& page = blocks[blk].pages[offset];

// 유효하지 않은 페이지는 읽을 수 없음

if (page.state != VALID) return;

// 읽기 결과 출력

std::cout << "Read LPN " << logicalPage << " -> PPN " << ppn << ": " << page.data << "\n";

}

// CostBenefitFTL의 가비지 컬렉션 구현

void CostBenefitFTL::garbageCollect() {

// 이미 GC가 진행 중이면 중복 실행 방지

if (in\_gc) return;

in\_gc = true;

// 희생자 블록 선택을 위한 변수들

int victim\_block = -1;

double best\_score = -1.0;

const int current\_time = total\_logical\_writes;

const int recent\_threshold = 5; // 최근 쓰기 임계값

// 모든 블록을 순회하면서 cost-benefit 분석

for (int i = 0; i < total\_blocks; ++i) {

Block& blk = blocks[i];

// 이미 비어있는 블록은 건너뛰기

if (blk.is\_free) continue;

// 과제 조건: valid 페이지가 2개 이하인 블록만 GC 대상

if (blk.valid\_page\_cnt > 2) continue;

// 최근에 쓰여진 블록은 GC 대상에서 제외 (hot data 보호)

if (current\_time - blk.last\_write\_time < recent\_threshold) continue;

// Cost: 이동해야 할 유효 페이지 수

double cost = static\_cast<double>(blk.valid\_page\_cnt);

// Benefit: 해제할 수 있는 무효 페이지 수

double benefit = static\_cast<double>(blk.invalid\_page\_cnt);

// cost가 0이면 분모가 0이 되므로 건너뛰기

if (cost == 0.0) continue;

// Cost-Benefit 점수 계산 (benefit/cost 비율)

double score = benefit / cost;

// 가장 높은 점수의 블록을 희생자로 선택

if (score > best\_score) {

best\_score = score;

victim\_block = i;

}

}

// 적절한 희생자 블록을 찾지 못한 경우 GC 종료

if (victim\_block == -1) {

in\_gc = false;

return;

}

// 선택된 블록의 GC 수행 횟수 증가

blocks[victim\_block].gc\_cnt++;

// 희생자 블록의 모든 페이지를 순회

for (int i = 0; i < block\_size; ++i) {

Page& page = blocks[victim\_block].pages[i];

// 유효한 페이지만 다른 곳으로 이동

if (page.state == VALID) {

// 페이지의 논리 주소와 데이터 백업

int lpn = page.logical\_page\_num;

int data = page.data;

// 기존 물리 주소 백업하고 L2P 테이블에서 임시 제거

int prev\_ppn = L2P[lpn];

L2P[lpn] = -1;

// GC 중이라는 플래그 유지

bool prev\_gc = in\_gc;

in\_gc = true;

// 유효한 데이터를 새로운 위치에 쓰기

writePage(lpn, data);

// 롤백: writePage 실패 시 복원

if (L2P[lpn] == -1) {

// 원래 페이지 복구

page.state = VALID;

blocks[victim\_block].valid\_page\_cnt++;

blocks[victim\_block].invalid\_page\_cnt--;

// L2P 테이블도 원래 상태로 복구

L2P[lpn] = prev\_ppn;

}

// GC 플래그 복원

in\_gc = prev\_gc;

// WAF 통계에서 GC로 인한 쓰기는 제외 (정상 쓰기만 카운트)

if (L2P[lpn] != -1)

total\_logical\_writes--;

}

}

// 블록 초기화 전에 GC 횟수 백업

int gc\_count = blocks[victim\_block].gc\_cnt;

// 블록을 완전히 초기화

blocks[victim\_block] = Block();

// GC 횟수 복원

blocks[victim\_block].gc\_cnt = gc\_count;

// 페이지 배열 재할당

blocks[victim\_block].pages.resize(block\_size);

// GC 완료 플래그 해제

in\_gc = false;

}

// CostBenefitFTL의 페이지 쓰기 구현

void CostBenefitFTL::writePage(int logicalPage, int data) {

// 논리 페이지 번호 유효성 검사

if (logicalPage < 0 || logicalPage >= static\_cast<int>(L2P.size())) return;

// 총 논리적 쓰기 횟수 증가

total\_logical\_writes++;

// GC 중이 아니고 GC 트리거 조건을 만족하면 GC 수행

if (!in\_gc && shouldTriggerGC()) {

garbageCollect();

}

// 기존에 매핑된 물리 페이지가 있으면 무효화

int old\_ppn = L2P[logicalPage];

if (old\_ppn != -1) {

// 물리 주소에서 블록 번호와 오프셋 계산

int old\_block = old\_ppn / block\_size;

int old\_offset = old\_ppn % block\_size;

Page& old\_page = blocks[old\_block].pages[old\_offset];

// 유효한 페이지를 무효로 변경

if (old\_page.state == VALID) {

old\_page.state = INVALID;

blocks[old\_block].valid\_page\_cnt--;

blocks[old\_block].invalid\_page\_cnt++;

}

}

// 블록이 없거나 꽉 찼으면 새로 할당

if (active\_block == -1 || active\_offset == block\_size) {

bool found = false;

// 모든 블록을 순회하면서 자유 블록 찾기

for (int i = 0; i < total\_blocks; ++i) {

if (blocks[i].is\_free) {

// 새 활성 블록 설정

active\_block = i;

active\_offset = 0;

// 블록을 사용 중으로 표시

blocks[i].is\_free = false;

// 페이지 배열 초기화

blocks[i].pages.resize(block\_size);

found = true;

break;

}

}

// 자유 블록을 찾지 못하고 GC 중이 아니면 GC 수행

if (!found && !in\_gc) {

garbageCollect();

// GC 후 다시 자유 블록 찾기

for (int i = 0; i < total\_blocks; ++i) {

if (blocks[i].is\_free) {

active\_block = i;

active\_offset = 0;

blocks[i].is\_free = false;

blocks[i].pages.resize(block\_size);

found = true;

break;

}

}

}

// 여전히 못 찾으면 write 실패

if (!found) return;

}

// 새 페이지에 데이터 쓰기

int ppn = active\_block \* block\_size + active\_offset;

Page& new\_page = blocks[active\_block].pages[active\_offset];

// 페이지 정보 설정

new\_page.logical\_page\_num = logicalPage;

new\_page.state = VALID;

new\_page.data = data;

// 블록의 유효 페이지 수 증가

blocks[active\_block].valid\_page\_cnt++;

// 마지막 쓰기 시간 기록

blocks[active\_block].last\_write\_time = total\_logical\_writes;

// L2P 테이블 업데이트

L2P[logicalPage] = ppn;

// 다음 쓰기를 위한 오프셋 증가

active\_offset++;

// 총 물리적 쓰기 횟수 증가

total\_physical\_writes++;

}

// CostBenefitFTL의 페이지 읽기 구현

void CostBenefitFTL::readPage(int logicalPage) {

// 논리 페이지 번호 유효성 검사

if (logicalPage < 0 || logicalPage >= static\_cast<int>(L2P.size())) return;

// L2P 테이블에서 물리 주소 찾기

int ppn = L2P[logicalPage];

// 매핑되지 않은 페이지는 읽을 수 없음

if (ppn == -1) return;

// 물리 주소에서 블록과 오프셋 계산

int blk = ppn / block\_size;

int offset = ppn % block\_size;

// 해당 페이지 참조

Page& page = blocks[blk].pages[offset];

// 유효하지 않은 페이지는 읽을 수 없음

if (page.state != VALID) return;

// 읽기 결과 출력

std::cout << "Read LPN " << logicalPage << " -> PPN " << ppn << ": " << page.data << "\n";

}

**전체 실행 결과**

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 영수증, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 영수증, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 번호, 라인이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

**보고서(구현 내용 설명, 문제 풀이, Discussion)**

**구현 내용 설명**

**ftl\_impl.h**

**멤버 변수 추가: bool in\_gc**

GC(Garbage Collection)가 진행 중인지 여부를 나타내는 플래그 변수이다.

**추가한 이유:**  
writePage 함수 내부에서 GC가 수행되는데, GC 내부에서도 유효 페이지를 새로운 위치에 쓰기 위해 다시 writePage를 호출한다. 이때 다시 GC 조건을 만족하게 되면 재귀적으로 GC가 호출될 수 있다. 이런 상황은 무한 루프 또는 스택 오버플로우(segmentation fault)로 이어질 수 있다.  
이를 방지하기 위해 in\_gc 플래그를 사용하여 현재 GC가 수행 중일 때는 추가적인 GC 호출을 막는다.

**사용 방식:**  
garbageCollect 함수 시작 시 in\_gc를 true로 설정하고, 종료 시 false로 되돌린다.

**멤버 함수 추가:**

**bool shouldTriggerGC()**

**역할:**  
현재 사용 중인 블록들 중에서 valid 페이지가 2개 이하인 블록이 존재하는지 확인하는 함수이다.

**추가한 이유:**  
과제 명세에 따라 GC는 valid 페이지가 2개 이하인 블록이 존재할 때만 수행해야 한다. 따라서 이를 판단하는 조건 검사 함수가 필요했다.

**사용 방식:**  
writePage 내부에서 GC 조건을 체크할 때 사용된다. 모든 블록을 순회하며 is\_free가 false이고 valid\_page\_cnt가 2 이하인 블록이 하나라도 존재하면 true를 반환한다.

**int getFreeBlockIndex()**

**역할:**  
자유 블록(free block) 중에서 가장 먼저 발견되는 블록의 인덱스를 반환한다.

**추가한 이유:**  
새로운 페이지를 쓰기 위해 빈 블록이 필요한데, 어떤 블록이 비어 있는지를 찾는 반복 코드가 writePage 내부에서 자주 사용되므로 이를 함수로 분리하였다.

**사용 방식:**  
0번 블록부터 순차적으로 순회하면서 is\_free가 true인 블록의 인덱스를 반환하고, 없으면 -1을 반환한다. writePage에서 active\_block을 새로 할당할 때 사용된다.

**bool hasVictimBlock()**

**역할:**  
GC가 수행 가능한 희생자 블록이 존재하는지를 판단하는 함수이다.

**추가한 이유:**  
GC를 수행할 수 있는 조건을 보다 명확히 하기 위해, 희생자 블록의 존재 여부를 사전에 판단해야 했다. 조건을 만족하지 않는 상황에서 불필요하게 garbageCollect를 호출하는 것을 방지하기 위함이다.

**사용 방식:**  
writePage 함수에서 shouldTriggerGC와 함께 사용되어 GC 실행 여부를 결정한다. 모든 블록을 순회하면서 is\_free가 false이고 valid\_page\_cnt가 2 이하이며 invalid\_page\_cnt가 1 이상인 블록이 있으면 true를 반환한다.

**ftl\_impl.cpp**

**void GreedyFTL::garbageCollect()**

* GC 중복 호출을 방지하기 위해 in\_gc를 true로 설정하고, 종료 시 false로 해제한다.
* victim\_block은 -1로 초기화되며, GC 대상이 없으면 그대로 종료된다.
* 블록 전체를 순회하면서 다음 조건을 만족하는 블록을 GC 후보로 탐색한다:
  1. is\_free == false인 블록만 고려
  2. valid\_page\_cnt > 2 이면 제외
  3. invalid\_page\_cnt > max\_invalid이면 victim\_block을 갱신
* 유효 페이지가 남아 있는 victim\_block의 각 페이지를 순회하면서, 상태가 VALID인 경우 다음을 수행:
  1. 페이지의 논리 주소와 데이터를 백업한다.
  2. 기존 L2P 매핑을 -1로 제거한 뒤, writePage를 호출해 유효 데이터를 다른 블록으로 옮긴다.
  3. writePage 실패 시 해당 페이지 상태와 L2P 테이블을 롤백한다.
  4. writePage가 성공했다면 WAF 계산을 보정하기 위해 total\_logical\_writes를 1 줄인다.
* GC 완료 후 victim\_block을 Block()으로 초기화한 뒤 gc\_cnt는 다시 복원하고, 페이지 배열은 resize로 재할당한다.
* 모든 GC 작업이 끝나면 in\_gc를 false로 설정하여 종료한다.

**void GreedyFTL::writePage(int logicalPage, int data)**

* 논리 페이지 번호가 유효하지 않으면 아무 작업도 하지 않고 종료한다.
* total\_logical\_writes를 1 증가시켜 총 논리 쓰기 횟수를 기록한다.
* 현재 GC 중이 아니고, shouldTriggerGC와 hasVictimBlock이 모두 true인 경우 garbageCollect를 수행한다.
* 논리 페이지가 이미 매핑되어 있는 경우, 이전 물리 페이지의 상태를 INVALID로 바꾸고 해당 블록의 valid/invalid 카운트를 갱신한다.
* 현재 활성 블록이 없거나 가득 찼을 경우 다음을 수행:
  + getFreeBlockIndex를 사용하여 자유 블록을 찾는다.
  + 없다면 GC를 수행하고 다시 getFreeBlockIndex를 호출한다.
  + 여전히 없으면 write 실패로 간주하고 total\_logical\_writes를 복구하고 종료한다.
* 사용할 수 있는 블록이 확보되면 다음 페이지에 데이터를 쓰고, 논리 주소를 기록한다:
  + ppn 계산 후 Page 객체에 논리 번호, 데이터, VALID 상태를 기록한다.
  + 블록의 valid\_page\_cnt를 1 증가시키고, 마지막 쓰기 시간으로 total\_logical\_writes 값을 기록한다.
  + L2P 테이블에 새로운 매핑을 저장하고, active\_offset을 1 증가시킨다.
  + total\_physical\_writes도 1 증가시켜 기록한다.

**void GreedyFTL::readPage(int logicalPage)**

* 논리 페이지 번호가 유효 범위를 벗어나면 종료한다.
* L2P 테이블에서 매핑된 물리 페이지 번호를 조회하고, -1이면 종료한다.
* 물리 주소로부터 블록 번호와 오프셋을 계산한다.
* 계산된 위치가 유효하지 않으면 종료한다.
* 해당 페이지의 상태가 VALID인 경우에만 데이터 값을 출력한다.
* 유효하지 않은 경우(예: INVALID 상태)에는 출력 없이 종료한다.

**void CostBenefitFTL::garbageCollect()**

Cost-Benefit 점수가 가장 높은 블록을 선택해 가비지 컬렉션을 수행한다.

* in\_gc 플래그로 중복 GC 호출을 방지한다.
* victim\_block과 best\_score를 초기화하고, 현재 시간인 total\_logical\_writes 값을 저장한다.
* 블록 전체를 순회하며 다음 조건에 따라 GC 대상을 필터링한다:
  1. is\_free가 true이면 이미 비어 있는 블록이므로 무시
  2. valid\_page\_cnt > 2이면 유효 페이지가 많아서 GC 대상이 아님
  3. current\_time - last\_write\_time < recent\_threshold 이면 최근 쓰기된 블록이므로 제외 (Hot data 보호)
* 유효한 블록에 대해 다음 점수를 계산함:
  1. cost = valid\_page\_cnt (이동해야 할 유효 페이지 수)
  2. benefit = invalid\_page\_cnt (얻을 수 있는 공간)
  3. score = benefit / cost
* 점수가 가장 높은 블록을 victim\_block으로 선택한다.
* 만약 victim\_block이 -1이라면, GC를 수행할 수 없으므로 종료한다.
* 이후 선택된 블록 내 모든 페이지를 순회하며 유효 페이지를 writePage로 옮긴다. 옮기기 전에는 L2P를 -1로 설정하고, GC 실패 시 롤백을 위해 prev\_ppn을 백업한다.
* writePage 수행 후 실패하면 원래 상태를 복구한다.
* writePage가 성공하면 GC로 발생한 논리적 쓰기를 total\_logical\_writes에서 1 감소시킨다 (WAF 보정).
* GC가 끝나면 블록을 초기화하고, GC 횟수를 보존하며 재할당한다.
* in\_gc를 false로 돌려 GC 종료 상태로 만든다.

**void CostBenefitFTL::writePage(int logicalPage, int data)**

* 논리 페이지 번호가 범위를 벗어나면 아무 작업도 하지 않고 종료한다.
* total\_logical\_writes를 1 증가시켜 쓰기 횟수를 반영한다.
* shouldTriggerGC()가 true이고 in\_gc가 false이면 garbageCollect()를 수행한다.
* 현재 논리 페이지에 매핑된 물리 페이지가 있다면 다음을 수행:
  + 해당 물리 페이지의 블록과 오프셋을 계산하고
  + 해당 페이지가 VALID 상태면 INVALID로 변경한다.
  + 블록 메타 정보인 valid\_page\_cnt를 감소시키고, invalid\_page\_cnt를 증가시킨다.
* 현재 사용 중인 active\_block이 비어 있거나 페이지가 가득 찬 경우 새로운 블록을 할당한다.
  + total\_blocks 범위 내에서 is\_free인 블록을 순차 탐색하며 할당
  + 블록을 찾지 못했을 경우, GC를 수행한 뒤 다시 찾는다.
  + 찾지 못하면 write를 포기하고 종료한다.
* 활성 블록이 존재하면, 다음 작업을 수행:
  + 새로운 ppn을 계산하고, 해당 위치의 Page 객체에 논리 번호, 데이터, VALID 상태를 저장
  + 해당 블록의 valid\_page\_cnt를 증가시키고, 마지막 쓰기 시간을 total\_logical\_writes로 설정
  + L2P[logicalPage]에 새로운 ppn을 기록하고 active\_offset을 증가시킨다
  + 총 물리적 쓰기 횟수도 1 증가시킨다

**void CostBenefitFTL::readPage(int logicalPage)**

* 논리 페이지 번호 유효성 검사 후, L2P에서 매핑된 물리 페이지 번호(ppn)를 조회한다.
* ppn이 -1이면 매핑이 없는 것이므로 종료한다.
* 물리 주소를 통해 블록 번호와 오프셋을 계산하고, 유효 범위를 벗어나면 종료한다.
* 해당 위치의 페이지가 VALID 상태인 경우에만 데이터를 출력한다.
* 그 외 상태이면 아무것도 출력하지 않는다.

**문제풀이**

텍스트, 스크린샷, 폰트, 일렉트릭 블루이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트, 운영 체제이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 폰트, 스크린샷, 일렉트릭 블루이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트, 로고이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

**Discussion**

**1. FTL 정책에 따른 WAF 및 GC 횟수 변화**

Workload B의 경우, Greedy 정책은 유효 페이지가 2개 이하이면서 invalid 페이지가 가장 많은 블록을 우선 선택하여 GC를 수행한다. 이로 인해 단기적으로 가장 많은 공간을 확보할 수 있지만, 결과적으로 Write Amplification Factor(WAF)가 1.972로 높게 나타났으며, 블록당 GC 횟수도 28회 이상으로 과도한 경향을 보였다.  
반면, CostBenefit 정책은 유효 페이지 수와 시간 정보를 함께 고려하여 장기적인 이득이 큰 블록을 우선적으로 선택했다. 그 결과 WAF는 1.893로 상대적으로 낮아졌고, GC 분포도 전반적으로 균일하게 이루어졌다.

Workload C의 경우에도 비슷한 경향이 나타났으며, 특히 GreedyFTL은 WAF가 2.205로 매우 높게 측정되었고, 일부 블록에 GC가 집중되면서 과도한 wear-leveling 문제가 발생할 가능성이 나타났다. 반면, CostBenefitFTL은 1.300이라는 낮은 WAF를 기록하며 GC 효율이 높고, wear-leveling 면에서도 더 안정적인 결과를 보였다.

**2. 새롭게 배운 점 및 어려웠던 점**

유효 페이지가 적은 블록을 선택하는게 WAF 측면에서 반드시 유리하지 않다는 점을 이번 과제를 진행하면서 느꼈다. GC 정책이 장기적인 I/O 패턴을 고려해야 함을 체험할 수 있었다. 특히 CostBenefitFTL의 구현에서는 블록의 마지막 쓰기 시간, 유효 페이지 수, invalid 페이지 수를 종합적으로 고려하는 식을 설계하고 적용하는 과정이 까다로웠다.

또한 writePage() 내부에서 GC가 재귀적으로 발생하는 문제를 방지하기 위한 in\_gc 플래그 설계는 GC 중 GC 발생이라는 흔한 버그를 실제로 겪고 해결하는 계기가 되어, 재진입 방지 로직의 중요성을 느꼈다. 이를 통해 실질적인 메모리 오류(Segmentation fault)의 원인을 디버깅하고 해결하는 능력을 키울 수 있었다.

**3. 기타 느낀 점**

이번 과제를 통해 단순한 알고리즘 구현을 넘어서서, 실제 시스템에서의 자원 관리(메모리/스토리지)가 기준과 정책에 따라 달라질 수 있는지 경험할 수 있었다. 또한 L2P 테이블의 정확한 업데이트, 블록의 상태 추적, 유효 데이터의 마이그레이션 등 실제 FTL 내부의 많은 구조를 이해해 볼 수 있었다.